Alpha AI Intern Evaluation Tasks

Prepared by: Alpha AI Team
Purpose: To assess applicants' technical skills, creativity, and problem-solving for the **Full Stack Web Developer** and **React Native/Flutter App Developer** intern roles.

Instructions for Applicants

- **Task Completion**: Select the role you applied for and complete the tasks below. You may use AI tools (e.g., GitHub Copilot, ChatGPT) and official documentation, but your work must reflect your own creativity and reasoning.
- **Timeframe**: Submit within 2 days i.e. 48 hours from receiving the tasks, maximum by 11:59 PM IST on the second calendar day. Submissions received before the deadline will receive priority review.
- **Deliverables**:
    - A GitHub repository with your code and a detailed README.md explaining your approach, design decisions, and trade-offs.
    - Screenshots or a demo video (optional but encouraged).
- **Evaluation**: We value clean, modular code, logical reasoning, creativity, and innovative ideas. There's no single "correct" answer - focus on showcasing your thought process and enjoy the challenge!

Full Stack Web Developer - Intern Tasks

Task 1: Build a Dockerized Web Application

- **Description**: Develop a web application using **React** or **Flutter (web)** that interacts with a backend API to manage a list of tasks. The app should display tasks, allow users to add or remove tasks, and handle errors gracefully.
- **Requirements**:
    - Use **Docker** to containerize both the frontend and backend for end-to-end workflow testing.
    - Build a simple backend API with **Node.js (Express)** or **Python (Flask/Django)** to manage tasks.
    - Ensure the app is responsive and handles errors (e.g., network issues).
- **Sample Data Structure**:
    - Tasks have the following fields:
        - id: unique identifier (integer)
        - title: short task name (text, e.g., "Finish report")

        - description: task details (text, e.g., "Complete Q3 summary")
        - status: task state (text, e.g., "todo", "in progress", "done")
    - Example tasks:

JSON

```
[
  {"id": 1, "title": "Finish report", "description": "Complete Q3 summary", "status":
"todo"},
  {"id": 2, "title": "Team meeting", "description": "Discuss project goals",
"status": "done"}
]
```

- **What We're Looking For**:
    - Clean, modular code separating frontend and backend logic.
    - A working Docker setup for easy deployment.
    - Creative UI elements or API features.

Task 2: Add Basic Caching for Performance

- **Description**: Implement a caching mechanism to improve app performance by reducing redundant API calls.
- **What is Caching?**: Caching stores data (like your task list) locally in the browser, so it loads faster without always hitting the server.
- **How to Implement It**:
    1. On app load, check **local storage** for the task list (e.g., localStorage.getItem('tasks')).
    2. If found, use it; otherwise, fetch from the server and save it (e.g., localStorage.setItem('tasks', JSON.stringify(tasks))).
    3. Update both the server and local storage when adding or removing tasks.
- **What We're Looking For**:
    - A functional caching system.
    - A README explanation of your approach.
    - Optional: Creative additions, like a manual cache refresh option.
- **Example Code Snippet (JavaScript)**:

JAVASCRIPT

```javascript
// Load tasks with caching
function loadTasks() {
  const cachedTasks = localStorage.getItem('tasks');
  if (cachedTasks) {
    return JSON.parse(cachedTasks); // Use cached data
  } else {
    return fetchTasksFromServer().then(tasks => {
      localStorage.setItem('tasks', JSON.stringify(tasks)); // Save to cache
      return tasks;
    });
  }
}

// Update tasks and cache
function addTask(newTask) {
  return fetch('/api/tasks', { method: 'POST', body: JSON.stringify(newTask) })
    .then(response => response.json())
    .then(updatedTasks => {
      localStorage.setItem('tasks', JSON.stringify(updatedTasks)); // Update cache
      return updatedTasks;
    });
}
```

Task 3: Add a Vector Search Feature

- **Description**: Enhance your backend with **PostgreSQL** and **pgvector** to store vector embeddings of task descriptions, enabling a feature to find similar tasks.
- **What is Vector Search?**: Vector search identifies tasks with similar meanings, not just exact matches. For example, searching "buy groceries" could find tasks like "get milk" based on content similarity.
- **Database Schema**:
    - Add an embedding column to the tasks table:
        - embedding: vector (stores the vector representation of the description, using pgvector)
- **How to Generate Embeddings**:
    - Use the **Sentence Transformers** library (all-MiniLM-L6-v2 model) to convert description text into vectors.
    - Example in Python:

PYTHON

```python
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
embedding = model.encode("Milk, eggs, bread").tolist()
```

    - Store the embedding when a task is created or updated.

- **Implement Vector Search**:
  - Add a search feature where users enter a query (e.g., "shopping"), generate its embedding, and find the top 3 similar tasks.
  - Example SQL query:

SQL

```sql
SELECT id, title, description, status
FROM tasks
ORDER BY embedding <-> '[query_embedding_here]'
LIMIT 3;
```

- **What We're Looking For**:
  - Proper setup of pgvector and embedding storage.
  - A working search feature returning relevant tasks.
  - Creative presentation of search results (e.g., highlighting similarity).
- **Optional Challenge**:
  - Add offline support using service workers or IndexedDB to view cached tasks without internet.

React Native/Flutter App Developer - Intern Tasks

Task 1: Create an Offline Mobile App

- **Description**: Build a mobile app using **Flutter** or **React Native** to manage a task list, functioning offline with **SQLite**.
- **Requirements**:
  - Support adding, editing, and deleting tasks.
  - Use SQLite for local storage and handle offline changes.
  - Simulate syncing with a mock server when online, managing conflicts (e.g., edited tasks).
- **Sample Data Structure**:
  - Tasks have the following fields:
    - id: unique identifier (integer)
    - title: short task name (text, e.g., "Finish report")
    - description: task details (text, e.g., "Complete Q3 summary")
    - status: task state (text, e.g., "todo", "in progress", "done")
  - Example tasks:

JSON

```json
[
  {"id": 1, "title": "Finish report", "description": "Complete Q3 summary", "status": "todo"},
  {"id": 2, "title": "Team meeting", "description": "Discuss project goals", "status": "done"}
]
```

- **What We're Looking For**:
    - Modular code with effective state management.
    - Clear offline and syncing logic.
    - Creative UI or conflict-resolution ideas.

Task 2: Add Offline AI Functionality

- **Description**: Integrate an offline AI model to analyze task descriptions and show their sentiment (positive, negative, neutral).
- **What is Offline AI?**: A pre-trained model runs on the device, no internet required.
- **Database Schema** (SQLite):
    - Add a sentiment column to the tasks table:
        - sentiment: text (stores the result, e.g., "positive")
- **How to Implement It**:
    - Use **TensorFlow Lite** with a pre-trained sentiment model from TensorFlow Hub (e.g., "sentiment-analysis").
    - When a task is added, analyze the description and store the sentiment in the sentiment column.
    - Display the sentiment in the app.
- **Sample Inputs and Outputs**:
    - description: "I love this project!" → sentiment: "positive"
    - description: "This is really frustrating." → sentiment: "negative"
    - description: "Meeting at 3 PM." → sentiment: "neutral"
- **What We're Looking For**:
    - Successful model integration and sentiment storage.
    - Clear sentiment display.
    - Creative uses of sentiment (e.g., color-coding tasks).

Task 3: Generate and Test an APK

- **Description**: Create an Android APK for your app and provide testing instructions.
- **Requirements**:
  - Ensure offline functionality (task management, sentiment analysis) works.
  - Test smooth transitions to online mode with syncing.
- **What We're Looking For**:
  - A working APK with clear testing steps.
  - Reliable offline and online behavior.
  - Optional: A build script (e.g., GitHub Actions).
- **Optional Challenge (Attempt it only if you have time)**:
  - Dockerize a mock backend and include a docker-compose.yml for testing.

Submission Instructions

- **Deadline**: Refer to the **Timeframe**.
- **How to Submit**:
  - Upload to a GitHub repository with a detailed README.md.
  - Include screenshots or a demo video (optional).
  - Email the repository link to hroperations@alphaai.biz with the subject: "Intern Evaluation - [Your Name] - [Role]" or share a google drive link with the runnable code and instructions.

Evaluation Criteria (All Roles)

- **Code Cleanliness**: Organized, readable, modular code.
- **Logical Deductions**: Clear reasoning for your solutions.
- **Creativity and Innovation**: Unique ideas or approaches.
- **Documentation**: Detailed process and decision explanations.

Notes for Applicants

- Use AI tools and docs (e.g., Flutter, TensorFlow Lite) for guidance, but create original solutions.
- Build something you're proud of, there's no strict "right" way.
- Tackle tasks step-by-step and ensure you adhere to requirements and deliverables.